Using Bounded Model Checking to Focus Fixpoint Iterations

David Monniaux, Laure Gonnord

Using Bounded Model Checking to Focus Fixpoint Iterations *

David Monniaux † Laure Gonnord ‡

June 13, 2011

Abstract

Two classical sources of imprecision in static analysis by abstract interpretation are widening and merge operations. Merge operations can be done away by distinguishing paths, as in trace partitioning, at the expense of enumerating an exponential number of paths.

In this article, we describe how to avoid such systematic exploration by focusing on a single path at a time, designated by SMT-solving. Our method combines well with acceleration techniques, thus doing away with widenings as well in some cases. We illustrate it over the well-known domain of convex polyhedra.

1 Introduction

Program analysis aims at automatically checking that programs fit their specifications, explicit or not — e.g. "the program does not crash" is implicit. Program analysis is impossible unless at least one of the following holds: it is unsound (some violations of the specification are not detected), incomplete (some correct programs are rejected because spurious violations are detected), or the state space is finite (and not too large, so as to be enumerated explicitly or implicitly). *Abstract interpretation* is sound, but incomplete: it over-approximates the set of behaviours of the analysed program; if the over-approximated set contains incorrect behaviours that do not exist in the concrete program, then false alarms are produced. A central question in abstract interpretation is to reduce the number of false alarms, while keeping memory and time costs reasonable [8].

Our contribution is a method leveraging the improvements in SMT-solving to increase the precision of invariant generation by abstract fixpoint iterations. On practical examples from the literature and industry, it performs better than previous generic technique and is less "ad-hoc" than syntactic heuristics found in some pragmatic analyzers.

*This research was partially funded by ANR project "ASOPT".

†CNRS, VERIMAG, Gières, France

‡Université Lille 1, LIFL, Villeneuve d'Ascq, France

Listing 1: C implementation of $y = \sin(x)/x - 1$, with the $-0.01 \leq x \leq 0.01$ range implemented using a Taylor expansion around zero in order to avoid loss of precision and division by zero as $\sin(x) \simeq x \to 0$.

```
if (x >= 0) { xabs = x; } else { xabs = -x; }
if (xabs >= 0.01) {
  y = sin(x) / x - 1;
} else {
  xsq = x*x;  y = xsq*(-1/6. + xsq/120.);
}
```

The first source of imprecision in abstract interpretation is the choice of the set of properties represented inside the analyser (the *abstract domain*). Obviously, if the property to be proved cannot be reflected in the abstract domain (e.g. we wish to prove a numerical relation but our abstract domain only considers Boolean variables), then the analysis cannot prove it.

In order to prove that there cannot be a division by zero in the first branch of the second if-then-else of Listing 1, one would need the non-convex property that $x \geq 0.01 \vee x \leq -0.01$. An analysis representing the invariant at that point in a domain of convex properties (intervals, polyhedra, etc.) will fail to prove the absence of division by zero (incompleteness).

Obviously, we could represent such properties using disjunctions of convex polyhedra, but this leads to combinatorial explosion as the number of polyhedra grows: at some point heuristics are needed for merging polyhedra in order to limit their number; it is also unclear how to obtain good widening operators on such domains. The same expressive power can alternatively be obtained by considering all program paths separately ("merge over all paths") and analysing them independently of each other. In order to avoid combinatorial explosion, the *trace partitioning* approach [36] applies merging heuristics. In contrast, our method relies on the power of modern SMT-solving techniques.

The second source of imprecision is the use of *widening operators* [14]. When analysing loops, static analysis by abstract interpretation attempts to obtain an *inductive invariant* by computing an increasing sequence $X_1, X_2, \ldots$ of sets of states, which are supersets of the sets of states reachable in at most $1, 2, \ldots$ iterations. In order to enforce convergence within finite time, the most common method is to use a widening operator, which extrapolates the first iterates of the sequence to a candidate limit. Optional narrowing iterations may regain some precision lost by widening.

Illustrating Example Consider Listing 2, a simplification of a fragment of an actual industrial reactive program: indexing of a circular buffer used only at certain iterations of the main loop of the program, chosen non-deterministically. If the non-deterministic choice nondet() is replaced by **true**, analysis with widening and narrowing finds $[0, 99]$. Unfortunately, the "narrowing" trick is brittle, and on

Listing 2: Circular buffer indexing

```
int x = 0;
while (true) {
  if (nondet()) {
    x = x+1;
    if (x >= 100) x = 0;
} }
```

Listing 2, widening yields $[0, +\infty)$, and this is not improved by narrowing! [1] In contrast, our semantically-based method would compute the $[0, 99]$ invariant on this example by first *focusing* on the following path inside the loop:

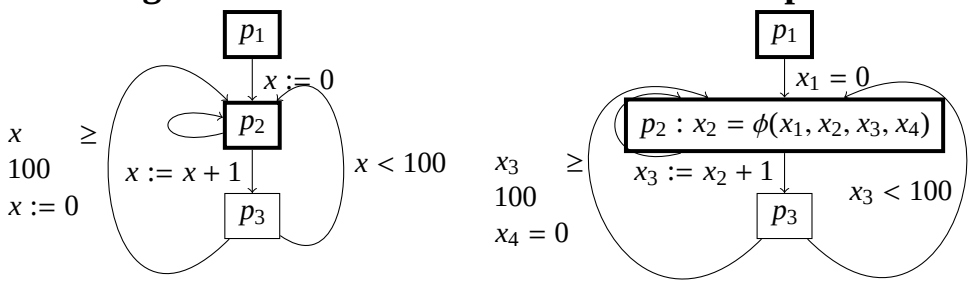
Listing 3: Example focus path

```
assume(nondet());  x = x+1;  assume(x < 100);
```

If we wrap this path inside a loop, then the least inductive invariant is $[0, 99]$. We then check that this invariant is inductive for the original loop.

This is the basic idea of our method: it performs fixpoint iterations by focusing temporarily on certain paths in the program. In order to obtain the next path, it performs bounded model checking using SMT-solving.

2 Background and Notations in Abstract Interpretation



(a) With original variables

(b) SSA version. $x = \phi(e_1, e_2, \dots)$ denotes a SSA ϕ-node: x takes value e_1 if control flows from the first incoming edge, e_2 from the second…

Figure 1: Control flow graph corresponding to listing 2.

We consider programs defined by a control flow graph: a set P of control points, for each control point $p \in P$ a (possibly empty) set I_p of initial values, a set $E \subseteq P \times P$ of directed edges, and the semantics $\tau_e : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ of each edge $e \in E$

[1] On this example, it is possible to compute the $[0, 99]$ invariant by so called "widening up-to" [28, Sec. 3.2], or with "thresholds" [8]: essentially, the analyser notices syntactically the comparison $x < 100$ and concludes that 99 is a "good value" for x, so instead of widening directly to $+\infty$, it first tries 99. This method only works if the interesting value is a syntactic constant.

where $\mathcal{P}(\Sigma)$ is the set of possible values of the tuple of program variables. τ_e thus maps a set of states before the transition expressed by edge e to the set of states after the transition.

To each control point $p \in P$ we attach a set $X_p \subseteq \Sigma$ of reachable values of the tuple of program variables at program point p. The concrete semantics of the program is the least solution of a system of semantic equations [14]: $X_p = I_p \cup \bigcup_{(p',p)\in E} \tau_{(p',p)}(X_{p'})$.

Abstract interpretation replaces the concrete sets of states in $\mathcal{P}(\Sigma)$ by elements of an abstract domain D. In lieu of applying exact operations τ to sets of concrete program states, we apply abstract counterparts $\tau^\sharp$.[2] An abstraction $\tau^\sharp$ of a concrete operation τ is deemed to be correct if it never "forgets" states:

$$\forall X \in D\ \tau(X) \subseteq \tau^\sharp(X) \tag{1}$$

We also assume an "abstract union" operation $\sqcup$, such that $X \cup Y \subseteq X \sqcup Y$. For instance, Σ can be $\mathbb{Q}^n$, D can be the set of convex polyhedra and $\sqcup$ the convex hull operation [27, 17, 3].

In order to find an inductive invariant, one solves a system of abstract semantic inequalities:

$$\begin{cases} \forall p\ I_p \subseteq X_p \\ \forall (p',p) \in E\ \tau^\sharp_{(p',p)}(X_{p'}) \subseteq X_p. \end{cases} \tag{2}$$

Since the $\tau^\sharp_e$ are correct abstractions, it follows that any solution of such a system defines an inductive invariant; one wishes to obtain one that is as strong as possible ("strong" meaning "small with respect to $\subseteq$"), or at least sufficiently strong as to imply the desired properties.

Assuming that all functions $\tau^\sharp_e$ are monotonic with respect to $\subseteq$, and that $\sqcup$ is the least upper bound operation in D with respect to $\subseteq$, one obtains a system of monotonic abstract equations: $X_p = I_p \sqcup \bigsqcup_{(p',p)\in E} \tau^\sharp_{(p',p)}(X_{p'})$. If $(D, \subseteq)$ has no infinite ascending sequences ($d_1 \subsetneq d_2 \subsetneq \ldots$ with $d_1, d_2, \cdots \in D$), then one can solve such a system by iteratively replacing the contents of the variable on the left hand side by the value of the right hand side, until a fixed point is reached. The order in which equations are iterated does not change the final result.

Many interesting abstract domains, including that of convex polyhedra, have infinite ascending sequences. One then classically uses an extrapolation operator known as *widening* and denoted by $\triangledown$ in order to enforce convergence within finite time. The iterations then follow the "upward iteration scheme":

$$X_p := X_p \triangledown \left(X_p \sqcup \bigsqcup_{(p',p)\in E} \tau^\sharp_{(p',p)}(X_{p'}) \right) \tag{3}$$

where the contents of the left hand side gets replaced by the value of the right hand side. The convergence property is that any sequence u_n of elements of D of the

[2] Many presentations of abstract interpretation distinguish the abstract element $x^\sharp \in D$ from the set of states $\gamma(x^\sharp)$ it represents. We opted not to, for the sake of brevity.

form $u_{n+1} = u_n \triangledown v_n$, where v_n is another sequence, is stationary [14]. It is sufficient to apply widening only at a set of program control nodes P_W such that all cycles in the control flow graph are cut. Then, through a process of *chaotic iterations* [13, Def. 4.1.2.0.5, p. 127], one converges within finite time to an inductive invariant satisfying Rel. 2.

Once an inductive invariant is found, it is possible to improve it by iterating the $\psi^\sharp$ function defined as $Y = \psi^\sharp(X)$, noting $X = (X_p)_{p\in P}$ and $Y = (Y_p)_{p\in P}$, with $Y_p = I_p \sqcup \bigsqcup_{(p',p)\in E} \tau^\sharp_{(p',p)}(X_{p'})$. If X is an inductive invariant, then for any k, $\psi^{\sharp k}(X)$ is also an invariant. This technique is an instance of *narrowing iterations*, which may help recover some of the imprecision induced by widening [14, §4].

Algorithm 1 Classical Algorithm

$A \leftarrow \emptyset$;
for all $p \in P$ such that $I_p \neq \emptyset$ **do**
    $A \leftarrow A \cup \{p\}$
end for; ▷ Initialise A to the set of all non empty initial nodes
while A is not empty **do** ▷ Fixpoint Iteration
    Choose $p_1 \in A$
    $A \leftarrow A \setminus \{p_1\}$
    for all outgoing edge (e) from p_1 **do**
        Let p_2 be the destination of e :
        if $p_2 \in P_W$ **then**
            $X_{temp} \leftarrow X_{p_2} \triangledown (X_{p_2} \sqcup \tau^\sharp_e(X_{p_1}))$ ▷ Widening node;
        else
            $X_{temp} \leftarrow X_{p_2} \sqcup \tau^\sharp_e(X_{p_1})$;
        end if
        if $X_{temp} \not\subseteq X_{p_2}$ **then** ▷ The value must be updated
            $X_{p_2} \leftarrow X_{temp}$;
            $A \leftarrow A \cup \{p_2\}$;
        end if
    end for;
end while; ▷ End of Iteration
Possibly narrow
return all X_{p_i}s;

A naive implementation of the upward iteration scheme described above is to maintain a work-list of program points p such that X_p has recently been updated and replaced by a strictly larger value (with respect to $\subseteq$), pick and remove the foremost member p, apply the corresponding rule $X_p := \ldots$, and insert into the work-list all p' such that $(p, p') \in E$ (This algorithm is formally described in Algorithm 1).

Example of Section 1 (Cont'd) Figure 1(a) gives the control flow graph obtained by compilation of Listing 2. Node p_2 is the unique widening node.

The classical algorithm (with the interval abstract domain) performs on this control flow graph of the following iterations :

- Initialisation : $X_{p_1} \leftarrow (-\infty, +\infty)$, $X_{p_2} \leftarrow X_{p_3} \leftarrow X_{p_4} \leftarrow \emptyset$.
- Step 1: $X_{p_2} \leftarrow [0, 0]$, then the transition to p_3 is enabled, $X_{p_3} \leftarrow [1, 1]$, then the return edge to p_2 gives the new point $x = 1$ to X_{p_2}, the new polyhedron is then $X_{p_2} = [0, 1]$ after performing the convex hull. Widening gives the polyhedron $X_{p_2} = [0, \infty)$.

  (The widening operator on intervals is defined as $[x_l, x_r] \triangledown [x'_l, x'_r] = [x"_l, x"_r]$ where $x"_l = x_l$ if $x_l = x'_l$ else $-\infty$, and $x"_r = x_r$ if $x_r = x'_r$ else $+\infty$.)
- Step 2: X_{p_3} becomes $[1, +\infty)$. The second transition from p_3 to p_2 is thus enabled, and the back edge to p_2 gives the point $x = 0$ to X_{p_2}. At the end of step 2 the convergence is reached.
- If we perform a narrowing sequence, there is no gain of precision because of the simple loop over the control point p_2.

3 Our Method

We have seen two examples of programs where classical polyhedral analysis fails to compute good invariants. How could we improve on these results?

- In order to get rid of the imprecision in Listing 1, one could "explode" the control-flow graph: in lieu of a sequence of n if-then-else, with n merge nodes with 2 input edges, one could distinguish the 2^n program paths, and having a single merge node with 2^n input edges. As already pointed out, this would lead to exponential blowup in both time and space.
- One way to get rid of imprecision of classical analysis (Sec. 2) on the program from Fig. 1(a) would be to consider each path through the loop at a time and compute a local invariant for this path. Again, the number of such paths could be exponential in the number of tests inside the loop.

The contribution of our article is a generic method that addresses both of these difficulties.

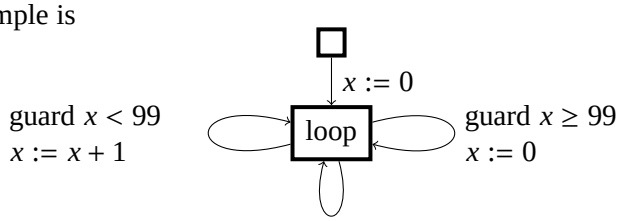
3.1 Reduced Transition Multigraph and Path Focusing

Consider a control flow graph (P, E) with associated transitions $(\tau_e)_{e \in E}$, a set of *widening points* $P_W \subseteq P$ such that removing P_W cuts all cycles in the graph, and a set P_R of *abstraction points*, such that $P_W \subseteq P_R \subseteq P$ (On the figures, the nodes in P_R are in bold). We make no assumption regarding the choice of P_W; there are

classical methods for choosing widening points [9, §3.6]. P_R can be taken equal to P_W, or may include other nodes; this makes sense only if these nodes have several incoming edges. Including other nodes will tend to reduce precision, but may improve scalability. We also make the simplifying assumption that the set of initial values I_p is empty for all nodes in $P \setminus P_R$ — in other words, the set of possible control points at program start-up is included in P_R.

We construct (virtually) the reduced control multigraph (P_R, E_R), with edges E_R consisting of the paths in (P, E) that start and finish on nodes in P_R, with associated semantics the composition of the semantics of the original edges $\tau_{e_1 \to \cdots \to e_n} = \tau_{e_n} \circ \cdots \circ \tau_{e_1}$. There are only a finite number of such edges, because the original graph is finite and removing P_R cuts all cycles. There may be several edges between two given nodes, because there may exist several control paths between these nodes in the original program. Equivalently, this multigraph can be obtained by starting from the original graph (P, E) and by removing all nodes p in $P \setminus P_R$ as follows: each couple of edges e_1, from p_1 to p, and e_2, from p to p_2, is replaced by a single edge from p_1 to p_2 with semantics $\tau_{p_2} \circ \tau_{p_1}$.

Example of Section 1 (Cont'd) The reduced control flow graph obtained for our running example is



Our analysis algorithm performs chaotic iterations over that reduced multigraph, without ever constructing it explicitly. We start from an iteration strategy, that is, a method for choosing which of the equations to apply next; one may for instance take a variant of the naive "breadth-first" algorithm from §2, but any iteration strategy [9, §3.7] befits us (see also Alg. 1). An iteration strategy maintains a set of "active nodes", which initially contains all nodes p such that $I_p \neq \emptyset$. It picks one edge e from an active node p_1 to a node p_2, and applies $X_{p_2} := X_{p_2} \sqcup \tau_e^\sharp(X_{p_1})$ in the case of a node $p_2 \in P_R \setminus P_W$, and applies $X_{p_2} := X_{p_2} \triangledown (X_{p_2} \sqcup \tau_e^\sharp(X_{p_1}))$ if $p_2 \in P_W$; then p_2 is added to the set of active nodes if the value of X_{p_2} has changed.

Our alteration to this algorithm is that we only pick edges e from p_1 to p_2 such that there exist $x_1 \in X_{p_1}$, $x_2 \in \tau_e(\{x_1\})$ and $x_2 \notin X_{p_2}$ with the current values of X_{p_1} and X_{p_2}. In other words, going back to the original control flow graph, we only pick paths that add new reachable states to their end node, and we temporarily *focus* on such a path.

How do we find such edges e out of potentially exponentially many? We express them as the solution of a *bounded reachability* problem — how can we go from control state p_1 with variable state in X_{p_1} to control state p_2 with variable state in X_{p_2} —, which we solve using satisfiability modulo theory (SMT). (See Alg. 2)

3.2 Finding Focus Paths

We now make the assumption that both the program transition semantics τ_e and the abstract elements $x^\sharp \in D$ can be expressed within a decidable theory T (this assumption may be relaxed by replacing the concrete semantics, including e.g. multiplicative arithmetic, by a more abstract one through e.g. linearization [30]).

Such is for instance the case if the program operates on rational values, so a program state is an element of $\Sigma = \mathbb{Q}^n$, all operations in the program, including guards and assignments, are linear arithmetic, and the abstract domain is the domain of convex polyhedra over $\mathbb{Q}^n$, in which case T can be the theory of linear real arithmetic (LRA). If program variables are integer, with program state space $\Sigma = \mathbb{Z}^n$, but still retaining the abstract domain of convex polyhedra over $\mathbb{Q}^n$, then we can take T to be the theory of linear integer arithmetic (LIA). Deciding the satisfiability of quantifier-free formulas in either LIA or LRA, with atoms consisting in propositional variables and in linear (in)equalities with integer coefficients, is NP-complete. There however exist efficient decision procedures for such formulas, known as SMT-solvers, as well as standardised theories and file formats [6]; notable examples of SMT-solvers capable of dealing with LIA and LRA are Z3 and Yices. Kroening & Strichman [29] give a good introduction to the techniques and algorithms in SMT solvers.

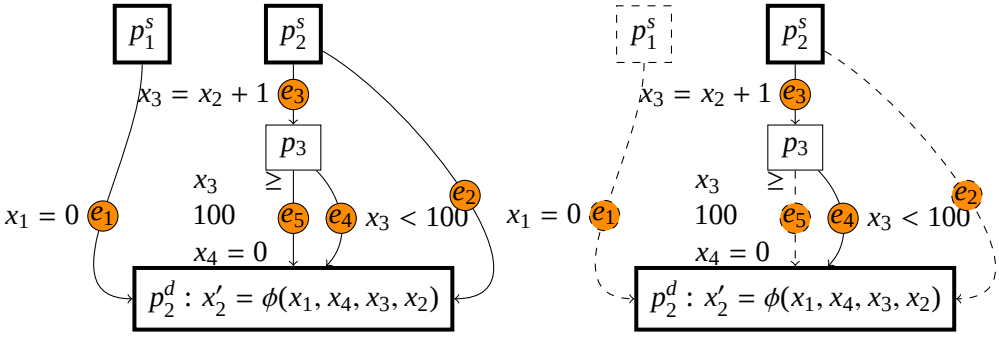
We assume that the program is expressed in SSA form, with each program variable being assigned a value at only a single point within the program [18]; standard techniques exist for converting to SSA. Figure 1 gives both "normal" and SSA-form control-flow graphs for Listing 2.

We transform the original control flow graph (P, E) in SSA form by disconnecting the nodes in P_R: each node p_r in P_R is split into a "source" node p_r^s with only outbound edges, and a "destination" node p_r^d with only inbound edges. We call the resulting graph (P', E'). Figure 2(a) gives the disconnected SSA form graph for Listing 2 where p_1 and p_2 have been split.

We consider execution traces starting from a p_r^s node and ending in a p_r^d node. We define them as for doing bounded model checking [2]. To each node $p \in P'$ we attach a Boolean b_p or *reachability predicate*, expressing that the trace goes through program point p. For nodes p' not of the form p_r^s, we have a constraint $b_{p'} = \bigvee_p e_{p,p'}$, for $e_{p,p'}$ ranging over all incoming edges. To each edge $p \to p'$ we attach a Boolean $e_{p,p'}$, and a constraint $e_{p,p'} = b_p \wedge \tau_{p,p'}$. The conjunction ρ of all these constraints, expresses the transition relation between the p_r^s and p_r^d nodes (with implicit existential quantification).

If the transitions $\tau_{(p,p')}$ are non-deterministic, a little care must be exercised for the path obtained from the b_p to be unique. For instance, if from program point p_1 one can move non-deterministically to p_2 or p_3 through edges e_2 and e_3 an incorrect way of writing the formula would be $(b_2 = e_2) \wedge (b_3 = e_3) \wedge (e_2 = b_1) \wedge (e_3 = b_1)$, in which case b_2 and b_3 could be simultaneously true. Instead, we introduce special "choice" variables c_i that model non-deterministic choices (Fig. 2).

In order to find a path from program point $p_1 \in P_R$, with variable state x_1,



(a) Disconnected (SSA) CFG

(b) With a focus path (solid edges) from $x_2 = 0$ at program point 2 to $x_2' = 1$ at the same program point

$(e_1 = (x_1 = 0) \wedge b_1^s) \wedge (e_3 = (x_3 = x_2 + 1) \wedge b_2^s \wedge c_2^s) \wedge (e_2 = b_2^s \wedge \neg c_2^s) \wedge (e_5 = b_3 \wedge x_3 \geq 100 \wedge x_4 = 0) \wedge (e_4 = b_3 \wedge x_3 < 100) \wedge (b_3 = e_3) \wedge (b_2^d = e_1 \vee e_4 \vee e_5 \vee e_2) \wedge (x_2' = ite(e_1, x_1, ite(e_5, x_4, ite(e_4, x_3, x_2))))$

Figure 2: Disconnected version of the SSA control flow graph of Fig. 1(b), and the corresponding SMT formula. $ite(b, e_1, e_2)$ is a SMT construct whose value is "if b then the value of e_1 else the value of e_2". To each node p_x corresponds a Boolean b_x and an optional choice variable c_x; to each edge, a Boolean e_y.

to program point $p_2 \in P_R$, with variable state x_2, we simply conjoin ρ with the formulas $x_1 \in X_{p_1}$ and $x_2 \notin X_{p_2}$, with x_1, x_2, $x_1 \in X_{p_1}$ and $x_2 \notin X_{p_2}$ expressed in terms of the SSA variables.[3] For instance, if X_{p_1} and X_{p_2} are convex polyhedra defined by systems of linear inequalities, one simply writes these inequalities using the names of the SSA-variables at program points p_1 and p_2.

We apply SMT-solving over that formula. The result is either "unsatisfiable", in which case there is no path from p_1, with variable values x_1, to p_2, with variable values x_2, such that $x_1 \in X_{p_1}$ and $x_2 \notin X_{p_2}$, or "satisfiable", in which case SMT-solving also provides a model of the formula (a satisfying assignment of its free variables); from this model we easily obtain such a path, unique by construction of ρ.

Indeed, a model of this formula yields a trace of execution: those b_p predicates that are true designate the program points through which the trace goes, and the other variables give the values of the program variables.

Example of Section 1 (Cont'd) The SSA form of the control flow graph of Figure 1(a) is depicted in Figure 1(b). Fig. 2 shows the disconnected version of the SSA Graph (the node p_2 is now split), and the formula ρ expressing the semantics is shown beneath it.

Then, consider the problem of finding a path starting in control point 2 inside polyhedron $x = 0$ and ending at the same control point but outside of that polyhe-

[3]The formula defining the set of values represented by an abstract element X has sometimes been denoted by $\hat{\gamma}$ [34].

dron. Note that because there are two outgoing transitions from node p_2^s, which are chosen non-deterministically, we had to introduce a Boolean choice variable c_2^s.

The focus path of Fig. 2(b) was obtained by solving the formula $\rho \wedge b_1^s = \text{false} \wedge b_2^s = \text{true} \wedge b_2^d = \text{true} \wedge (x_2 = 0) \wedge \neg(x_2' = 0)$: we impose that the path starts at point p_2^s (thus forcing $b_1^s = \text{false} \wedge b_2^s = \text{true}$) in the polyhedron $x = 0$ (thus $x_2 = 0$) and ends at point p_2^d (thus forcing $b_2^p = \text{true}$) outside of that polyhedron (thus $\neg(x_2 = 0)$).

3.3 Algorithm

Algorithm 2 consists in the iteration of the path finding method of Sec. 3.2, coupled with forward abstract interpretation along the paths found and, optionally, path acceleration.

3.4 Correctness and Termination

We shall now prove that this algorithm terminates, and that the resulting X_p define an inductive invariant that contains all initial states I_p. The proof is a variant of the correctness proof of the chaotic iterations.

The invariant maintained by this algorithm is that all nodes $p_1 \in P_R \setminus A$ are such that there is no execution trace starting at point p_1 in a state $x_1 \in X_{p_1}$ and ending at point p_2 in a state $x_2 \notin X_{p_2}$. Evidently, if A becomes empty, then this condition means that X_p is an inductive invariant.

Termination is ensured by the classical argument of termination of chaotic iterations in the presence of widening: they always terminate if all cycles in the control flow graph are broken by widening points [13, Th. 4.1.2.0.6, p. 128]. In short, an infinite iteration sequence is bound to select at least one node p in P_W an infinite amount of times, because P_W breaks all cycles, but due to the properties of widening, X_p should be stationary, which contradicts the infinite number of selections. Our comment at line 20 of Alg. 2 is important for termination: it ensures that for any widening node p, the sequence of values taken by X_p when it is updated and reinserted into set A is strictly ascending, which ensures termination in finite time.

3.5 Self-Loops

The algorithm in the preceding subsection is merely a "clever" implementation of standard polyhedral analysis [17, 27] on the reduced control multigraph (P_R, E_R); the difference with a naive implementation is that we do not have to explicitly enumerate an exponential number of paths and instead leave the choice of the focus path to the SMT-solver. We shall now describe an improvement in the case of self-loops, that is, single paths from one node to itself.

Algorithm 3 is a variant of Alg. 2 where self-loops are treated specially:

- The *loopiter*$(\tau^\sharp, X)$ function returns the result of a widening / narrowing iteration sequence for abstract transformer $\tau^\sharp$ starting in X; it returns X' such

Algorithm 2 Path-focused Algorithm

1: Compute SSA-form of the control flow graph.
2: Choose P_R, compute the disconnected graph (P', E') accordingly.
3: $\rho \leftarrow$ computeFormula(P', E') ▷ Precomputations
4: $A \leftarrow \emptyset$;
5: **for all** $p \in P_R$ such that $I_p \neq \emptyset$ **do**
6: $\quad A \leftarrow A \cup \{p\}$
7: **end for**;
8: **while** A is not empty **do** ▷ Fixpoint Iteration on the reduced graph
9: $\quad$ Choose $p_1 \in A$
10: $\quad A \leftarrow A \setminus \{p_1\}$
11: $\quad$ **repeat**
12: $\quad\quad res \leftarrow \text{SmtSolve}\left(\rho \wedge b_{p_1} \wedge x_1 \in X_{p_1} \wedge \bigvee_{p_2 | (p_1,p_2) \in E'} \left(b_{p_2} \wedge x_2 \notin X_{p_2}\right)\right)$
13: $\quad\quad$ **if** res is not "unsat" **then**
14: $\quad\quad\quad$ Compute $e' \in E'$ from res ▷ Extraction of path from the model (§3.2)
15: $\quad\quad\quad Y \leftarrow \tau^\sharp_{e'}(X_{p_1})$
16: $\quad\quad\quad$ **if** $p_2 \in P_W$ **then**
17: $\quad\quad\quad\quad X_{temp} \leftarrow X_{p_2} \triangledown (X_{p_2} \sqcup Y)$ ▷ Final point p_2 is a widening point
18: $\quad\quad\quad$ **else**
19: $\quad\quad\quad\quad X_{temp} \leftarrow X_{p_2} \sqcup Y$
20: $\quad\quad\quad$ **end if**
▷ at this point $X_{temp} \not\subseteq X_{p_2}$ otherwise p_2 would not have been chosen
21: $\quad\quad\quad X_{p_2} \leftarrow X_{temp}$
22: $\quad\quad\quad A \leftarrow A \cup \{p_2\}$
23: $\quad\quad$ **end if**
24: $\quad$ **until** res="unsat"
25: **end while** ▷ End of Iteration
26: Possibly narrow (see Sec. 4.1)
27: Compute X_{p_i} for $p_i \notin P_R$
28: **return** all X_{p_i}

that $X \subseteq X'$ and $\tau^\sharp(X') \subseteq X'$.

- In order not to waste the precision gained by *loopiter*, the first time we consider a self-loop e' we apply a union operation instead of a widening; set U records the self-loops that have already been visited. This is a form of delayed widening [28].

Termination is still guaranteed, because the inner loop cannot loop forever: it can visit any self-loop edge e' at most once before applying widening.

Algorithm 3 Path-focused Algorithm with Self-Loops. ■ marks changes from Alg. 2.

Compute SSA-form of the control flow graph.
Choose P_R, compute the disconnected graph (P', E') accordingly.
$\rho \leftarrow$ computeFormula(P', E') ▷ Precomputations
$A \leftarrow \emptyset$;
for all $p \in P_R$ such that $I_p \neq \emptyset$ **do**
    $A \leftarrow A \cup \{p\}$
end for;
while A is not empty **do** ▷ Fixpoint Iteration on the reduced graph
    Choose $p_1 \in A$
    $A \leftarrow A \setminus \{p_1\}$
    $U = \emptyset$ ▷ U is a set of "already seen" edges
    repeat
        $res \leftarrow \text{SmtSolve}\left(\rho \wedge b_{p_1} \wedge x_1 \in X_{p_1} \wedge \bigvee_{p_2 | (p_1,p_2) \in E'} \left(b_{p_2} \wedge x_2 \notin X_{p_2}\right)\right)$
        if res is not "unsat" **then**
            Compute $e' \in E'$ from res
            if $p_1 = p_2$ **then**
                $Y \leftarrow loopiter(\tau^\sharp_{e'}, X_{p_1})$
            else
                $Y \leftarrow \tau^\sharp_{e'}(X_{p_1})$
            end if
            if $p_2 \in P_W$ **and** $(p_1 \neq p_2 \vee e' \in U)$ **then**
                $X_{p_2} \leftarrow X_{p_2} \triangledown (X_{p_2} \sqcup Y)$ ▷ Final point p_2 is a widening point
            else
                $X_{p_2} \leftarrow X_{p_2} \sqcup Y$
                $U \leftarrow U \cup \{e'\}$
            end if
            $A \leftarrow A \cup \{p_2\}$
        end if
    until res="unsat"
end while ▷ End of Iteration
Compute X_{p_i}s for $p_i \notin P_R$
return all X_{p_i}s

Example of Section 1 (Cont'd) Let us perform our algorithm on our example :

- Step 1 : Is there a path from control point p_1 to control point p_2 feasible (without additional constraint) ? Yes. On Figure 2, the obtained model corresponds to the transition from p_1^s to p_2^d, and leads to the interval $X_{p_2} = [0, 0]$.
- Step 2 : Is there a path from p_2 with $x = 0$ to p_2 with $x \neq 0$? The answer to this query is depicted in Figure 2(b): there is such a path, on which we

now focus. This path is considered as a loop and we therefore do a local iteration with widenings (*loopiter*). X_{p_2} becomes $[0, 1]$, then after widening $[0, \infty]$. A narrowing step gives finally $X_{p_2} = [0, 99]$, which is thus the result of *loopiter*.

- Step 3 : Is there a path from p_2 with $x \in [0, 99]$ to p_2 with $x' \notin [0, 99]$? No.

The iteration thus ends with the desired invariant.

4 Extensions

4.1 Narrowing

Narrowing iterations can also be applied within our framework. Let us assume that some inductive invariant $X_{p \in P_R}$ has been computed; it satisfies the relation $\psi(X) \subseteq X$ component-wise, noting $X = (X_1, \ldots, X_{|P|})$, and $\psi(X)$ denotes $(Y_1, \ldots, Y_{|P|})$ defined as

$$Y_{p_2} = I_{p_2} \cup \bigcup_{e \in E_R \; e \text{ from } p_1 \text{ to } p_2} \tau_e\left(X_{p_1}\right) \tag{4}$$

The abstract counterpart to this operator is $\psi^\sharp$, defined similarly, replacing τ by $\tau^\sharp$ and $\cup$ by $\sqcup$. It satisfies the correctness condition (see Rel. 1) $\forall X \in D\, \psi(X) \subseteq \psi^\sharp(X)$.

As per the usual narrowing iterations, we compute a narrowing sequence $X^{(k)} = \psi^{\sharp k}(X)$. It is often sufficient to stop at $k = 1$; otherwise one may stop when $X^{(k+1)} \not\subseteq X^{(k)}$. Let us now see a practical algorithm for computing $Y = \psi^\sharp(X)$:

For all $p \in P_R$, we initialise $Y_p := I_p$. For all $p_2 \in P_R$, we consider all paths $e \in E_R$ from $p_1 \in P_R$ to p_2 such that there exist $x_1 \in X_{p_1}$, $x_2 \in X_{p_2}$, $x_2 \in \tau_e(\{x_1\})$ as explained in §3.2. We then update $Y_{p_2} := Y_{p_2} \sqcup \tau_e^\sharp(X_{p_1})$.

4.2 Acceleration

In Sec. 3.5, we have described *loopiter* function that performs a classical widening / narrowing iteration over a single path. In fact, the only requirement over it is that *loopiter*$(\tau^\sharp, X)$ returns X' such that $X \subseteq X'$ and $\tau^\sharp(X') \subseteq X'$. In other words, X' is an over-approximation of $\tau^{\sharp *}(X)$, noting R^* the transitive closure of R.

In some cases, we can compute directly such an over-approximation, sometimes even obtaining $\tau^{\sharp *}(X)$ exactly; this is known as *acceleration* of the loop. Examples of possible accelerations include the case where τ_e is given by a difference bound matrix [12], an octagon [10], ultimately periodic integer relations [11] or certain affine linear relations [23, 22, 1].

For instance, the focus path of Fig. 2(b) consists in the operations and guards $x = x + 1; x < 100$; instead of iterating that path, we can compute its exact acceleration, yielding $x \in [0, 99]$.

4.3 Partitioning

It is possible to partition the states at a given program point according to some predicate or a partial history of the computation [36]. This amounts to introducing several graph nodes representing the same program point, and altering the transition relation.

4.4 Input-Output Relations

As with other analyses using relational domains, it is possible to obtain abstractions of the input-output relation of a program block or procedure instead of an abstraction of the set of states at the current point [1]; this also allows analyzing recursive procedures [27, Sec. 7.2]. It suffices to include in the set of variables copies of the variables at the beginning of the block or procedure; then the abstract value obtained at the end of the block or procedure is the desired abstraction.

5 Implementation and Preliminary Results

Our algorithm has been implemented as an option for Aspic, that computes invariants from counter automata with Linear Relation Analysis ([20]). We wrote an Ocaml interface to the Yices SMT-solver ([19]), and modified the fixpoint computation inside Aspic to deal with local iterations of paths. The implementation still needs some improvements, but the preliminary results are promising, and we describe some of them in Table 1. We provide no timing results since we were unable to detect any overcost due to the method. These two examples show that since we avoid (some) convex hulls, the precision of the whole analysis is improved.

The rate limiter example is particularly interesting, since, like the one in Listing 1 (which does not include a loop), it will be imprecisely analyzed by any method enforcing convex invariants at intermediate steps.

6 Related Work

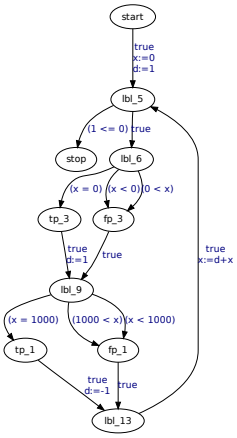
Our algorithm may be understood as a form of *chaotic iterations* [13, §2.9.1, p. 53] over a certain system of semantic questions; we use SMT as an oracle to know which equations need propagating. The choice of widening points, and the order in which to solve the abstract equations, have an impact on the precision of the whole analysis, as well as its running time. Even though there exist few hard general results as to which strategy is best [13, §4.1.2, p. 125], some methods tend to experimentally behave better [9].

"Lookahead widening" [24] was our main source of inspiration: iterations and widenings are adapted according to the discovery of new feasible paths in the program. This approach avoids loss of precision due to widening in programs with

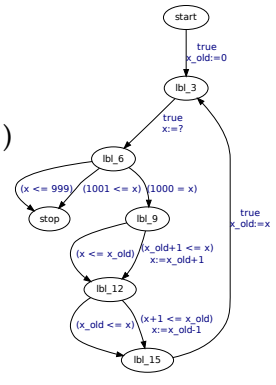
Table 1: Invariant generation on two simple challenging programs

<table>
<tr><th>Program</th><th>Automaton</th><th>Result and notes</th></tr>
<tr>
<td>Listing 4: Boustrophedon

```
void boustrophedon() {
  int x;
  int d;
  x = 0;
  d = 1;
  while (1) {
    if (x == 0) d=1;
    if (x == 1000) d=-1;
    x += d;
  }
}
```

</td>
<td> 
</td>
<td>The compilation of the program gives an expanded control structure where some paths are “clearly” unfeasible (e.g. imposing both $x < 0$ and $x > 1000$), thus the only feasible ones are guarded by $x < 0$, $x = 0$, $0 < x < 1000$, $x = 1000$ and $x > 1000$.
The tool finds the invariant $\{\mathbf{0 \leq x \leq 1000}, \mathbf{-1 \leq d \leq 1}\}$
Classical Analysis with widening “upto” gives $\{d \leq 1, d + 1999 \geq 2x\}$ and Gopan and Reps’ improvement is not able to find $x \geq 0$.</td>
</tr>
<tr>
<td>Listing 5: Rate limiter

```
void main() {
  float x_old, x;
  x_old = 0;
  while (1) {
    x = input(-1000,1000);
    if (x >= x_old+1)
        x = x_old+1;
    if (x <= x_old-1)
        x = x_old-1;
    x_old = x;
  }
}
```


Source : [32]</td>
<td> 
</td>
<td>In order to properly analyse such a program, Astrée distinguishes all four execution paths inside the loop through trace partitioning [36], which is triggered by ad hoc syntactic criteria (e.g. two successive if-then-else). Our algorithm finds the invariant $\{\mathbf{-1000 \leq x_{old} \leq 1000}\}$, which is not found by classical analysis.</td>
</tr>
</table>

multiple paths inside loops. It has proved its efficacy to suppress some gross over-approximations induced by naive widening. However, it does not solve the imprecisions introduced by convex hull (e.g. it produces false alarms on Listing 1).

Our method analyzes separately the paths between cut-nodes. We have pointed out that this is (almost) equivalent to considering finite unions of elements of the abstract domain, known as the *finite powerset* construction, between the cut-nodes.[4] The finite powerset construction is however costly even for loop-free code, and it is not so easy to come up with widening operators to apply it to codes with loops or recursive functions [4]; for limiting the number of elements in the unions, some may be lumped together (thus generally introducing further over-approximation) according to affinity heuristics [37, 33].

Still, in the recent years, much effort has been put into the discovery of *disjunctive invariants*, for instance in predicate abstraction [25]. Of particular note is the recent work by Gulwani and Zuleger on inferring disjunctive invariants [26] for finding bounds on the number of iterations of loops. We improve on their method on two points:

- In contrast to us, they assume that the transition relation is given in disjunctive normal form [26, Def. 5], which in general has exponential size in the number of tests inside the loop. By using SMT-solving, we keep the DNF implicit and thus avoid this blowup.

- By using acceleration, we may obtain more precise results than using widening, as they do for lattices that do not satisfy the ascending chain condition.

Nevertheless, their method allows expressing disjunctive invariants at loop heads, and not only at intermediate points, as we do. However, we think it is possible to get the best of both worlds and combine our method with theirs. In order to obtain a disjunctive invariant, they first choose a "convexity witness" (given that the number of possible witnesses is exponential, they choose it using heuristics) [26, p. 7], and then they compute a "transitive closure" [26, Fig. 6], which is a form of fixed point iteration of input-output relations (as in our Sec. 4.4) over an expanded control-flow graph. The choice of the convexity witness amounts to a partitioning of the nodes and transition (Sec. 4.3). Thus, it seems to possible to apply their technique, but replace their fixed point iteration [26, Fig. 6] by one based on SMT-solving and path focusing, using acceleration if possible.

In recent years, because of improvement in SMT-solving, techniques such as ours, distinguishing *paths* inside loops, have become tractable [31, 7, 32, 21]. An alternative to using SMT-solving is to limit the number and length of traces to consider, as in *trace partitioning* [36], used in the Astrée analyzer [16, 15, 8], but

[4] It is equivalent if the only source of disjunctions are the splits in the control flow, and not atomic operations. For instance, if the test $|x| \geq 1$ is considered an atomic operation, then we could take the disjunction $x \geq 1 \vee x \leq -1$ as output. We can rephrase that as a control flow problem by adding a test $x \geq 0$, otherwise said to express $|x|$ as a piecewise linear function with explicit tests for splits between the pieces.

the criteria for limitation tend to be ad hoc. In addition, methods for abstracting the sets of paths inside a loop, weeding out infeasible paths, have been introduced [5].

With respect to optimality of the results, our method will generate the strongest inductive invariant inside the abstract domain if the domain satisfies the ascending chain condition and no widening is used; for other domains, like all methods using widenings, it may or may not generate it. In contrast, some recent works [21] guarantee to obtain the strongest invariant for the same analysis problem, at the expense of restriction to template linear domains and linear constructions inside the code.

7 Conclusion and future work

We have described a technique which leverages the bounded model checking capacities of current SMT solvers for guiding the iterations of an abstract interpreter. Instead of normal iterations, which "push" abstract values along control-flow edges, including control-flow splits and merges, we consider individual paths. This enables us, for instance, to use acceleration techniques that are not available when the program fragment being considered contains control-flow merges. This technique computes exact least invariants on some examples on which more conventional static analyzers incur gross imprecision or have to resort to syntactic heuristics in order to conserve precision.

We have focused on numerical abstractions. Yet, one would like to use similar techniques for heap abstractions, for instance. The challenge will then be to use a decidable logic and an abstract domain such that both the semantics of the program statements and the abstract values can be expressed in this logic. This is one direction to explore. With respect to the partitioning technique, 4.3, we currently express the partition as multiple explicit control nodes, but it seems desirable, for large partitions (e.g. according to Boolean values, as in B. Jeannet's BDD-Apron library) to represent them succinctly; this seems to fit nicely with our succinct encoding of the transition relation as a SMT-formula.

Another direction is to evaluate the scalability of these methods on larger programs. The implementation needs to be tested more to evaluate the precision of our method on middle-sized programs, the main advantage is that Aspic implements some of the acceleration techniques. Analyzers such as Astrée scale up to programs running a control loop several hundreds of thousands of lines long; translating such a loop to a SMT formula and solving for this formula and additional constraints does not seem tractable. It is possible that semantic slicing techniques [35] could help in reducing the size of the generated SMT problems.

References

[1] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. *Electr. Notes Theor.*

Comput. Sci., 267(1):3–16, 2010. Proceedings of NSAD.

[2] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using smt solvers instead of sat solvers. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(1):69–83, 2009.

[3] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. *The Parma Polyhedra Library, version 0.9*.

[4] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Widening operators for powerset domains. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(4-5):449–466, August 2006. See also erratum in June 2007 issue.

[5] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivancic, and Aarti Gupta. Refining the control structure of loops using static analysis. In Samarjit Chakraborty and Nicolas Halbwachs, editors, *EMSOFT*, pages 49–58. ACM, 2009.

[6] Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (SMT-LIB). `www.smtlib.org`, 2008.

[7] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *PLDI*, pages 300–309. ACM, 2007.

[8] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation (PLDI)*, pages 196–207. ACM, 2003.

[9] François Bourdoncle. *Sémantique des langages impératifs d'ordre supérieur et interprétation abstraite*. PhD thesis, École polytechnique, Palaiseau, 1992.

[10] Marius Bozga, Codruta Gîrlea, and Radu Iosif. Iterating octagons. Technical Report 16, VERIMAG, 2008.

[11] Marius Bozga, Radu Iosif, and Filip Konecny. Fast acceleration of ultimately periodic relations. Technical Report 2010-3, VERIMAG, 2010.

[12] Hubert Comon and Yan Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-aided verification (CAV)*, volume 1427 of *LNCS*, pages 268–279. Springer, 1998.

[13] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes*. State doctorate thesis, Université scientifique et médicale de Grenoble and Institut National Polytechnique de Grenoble, 1978.

[14] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. of Logic and Computation*, pages 511–547, August 1992.

[15] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In Shmuel "Mooly" Sagiv, editor, *Programming Languages and Systems (ESOP)*, number 3444 in LNCS, pages 21–30. Springer, 2005.

[16] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science — ASIAN 2006*, volume 4435 of *LNCS*, pages 272–300. Springer, 2008.

[17] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL)*, pages 84–96. ACM, 1978.

[18] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Principles of programming languages (POPL)*, pages 25–35. ACM, 1989.

[19] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *Computer-aided verification (CAV)*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.

[20] Paul Feautrier and Laure Gonnord. Accelerated invariant generation for C programs with Aspic and C2fsm. In *Tools (TAPAS)*, 2010.

[21] Thomas Gawlitza and David Monniaux. Improving strategies via SMT solving. In *ESOP*, 2011. to appear.

[22] Laure Gonnord. *Accélération abstraite pour l'amélioration de la précision en analyse des relations linéaires*. PhD thesis, Université Joseph Fourier, October 2007.

[23] Laure Gonnord and Nicolas Halbwachs. Combining widening and acceleration in linear relation analysis. In Kwangkeun Yi, editor, *Static analysis (SAS)*, volume 4134 of *LNCS*, pages 144–160. Springer, 2006.

[24] Denis Gopan and Thomas W. Reps. Lookahead widening. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 452–466. Springer, 2006.

[25] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5403 of *LNCS*, pages 120–135. Springer, 2009.

[26] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI*, pages 292–304. ACM, 2010.

[27] Nicolas Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. State doctorate thesis, Université scientifique et médicale de Grenoble and Institut National Polytechnique de Grenoble, 1979.

[28] Nicolas Halbwachs. Delay analysis in synchronous programs. In Costas Courcoubetis, editor, *Computer Aided Verification (CAV)*, volume 697 of *LNCS*, pages 333–346. Springer, 1993.

[29] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer, 2008.

[30] Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *LNCS*, pages 348–363. Springer, January 2006.

[31] David Monniaux. Automatic modular abstractions for linear constraints. In Benjamin C. Pierce, editor, *Symposium on Principles of programming languages (POPL)*. ACM, 2009.

[32] David Monniaux. Automatic modular abstractions for template numerical constraints. *Logical Methods in Computer Science*, 2010. To appear.

[33] Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. In *ASIAN'06*, pages 331–345. Springer, 2007.

[34] Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *LNCS*, pages 252–266. Springer, 2004.

[35] Xavier Rival. Understanding the origin of alarms in Astrée. In Chris Hankin and Igor Siveroni, editors, *Static analysis (SAS)*, volume 3672 of *LNCS*, pages 303–319. Springer, 2005.

[36] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):26, 2007.

[37] Sriram Sankaranarayanan, Franjo Ivančić, Ilya Shlyahkter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In *Static analysis (SAS)*, volume 4134 of *LNCS*, pages 3–17. Springer, 2006.